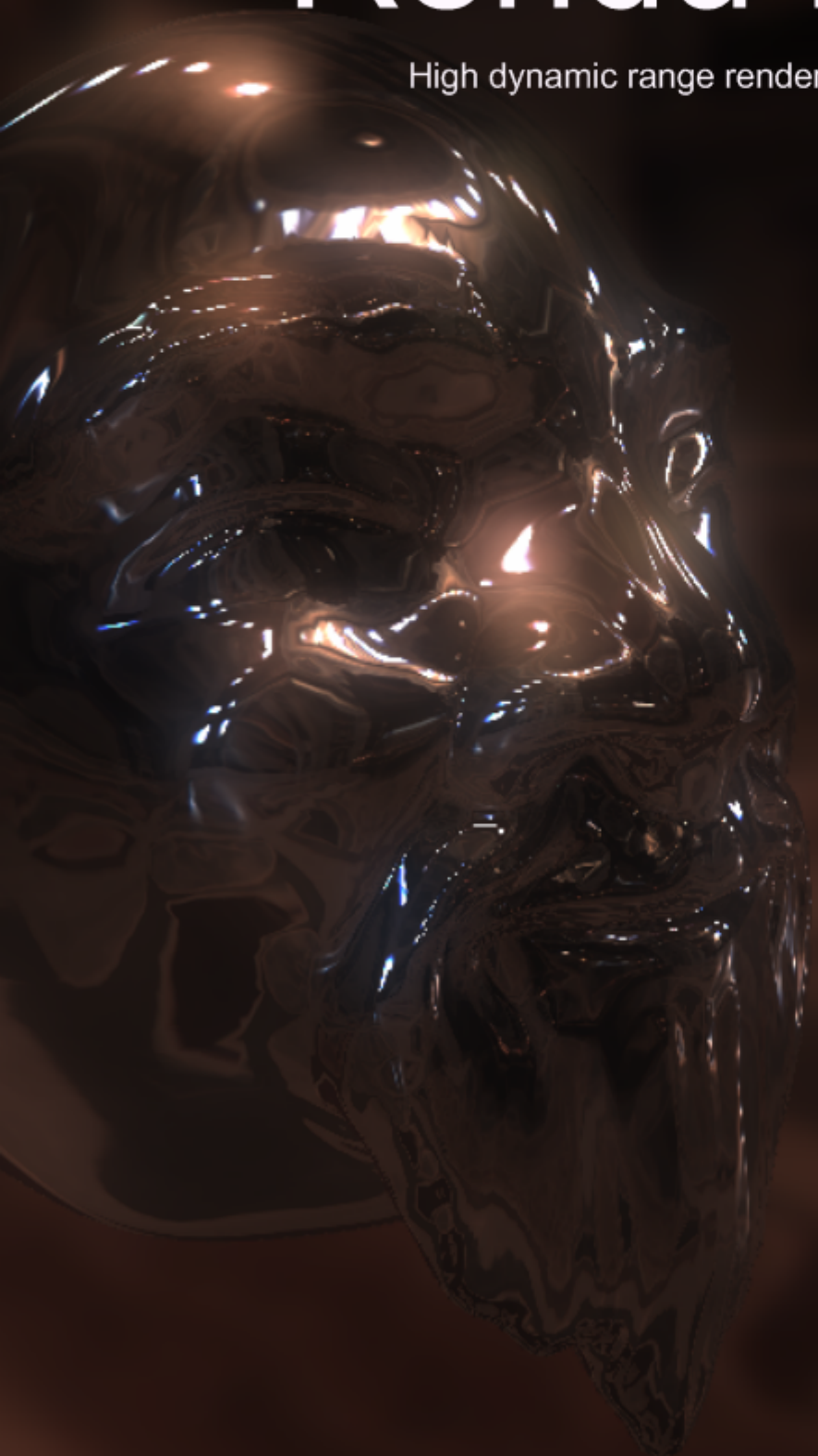
Alexandre Haffner
Nicolas Monneret
UTBM - Génie Informatique

IN55

Rendu HDR

High dynamic range rendering en OpenGL



Enseignant :
Mikael Goncalves

Choix du sujet

Dans le cadre de l'UV IN55 nous avons choisis de travailler le sujet portant sur le rendu HDR proposé par M. Goncalves. Nous avons arrêté ce choix pour de multiples raisons. Premièrement pour la nouveauté de la technologie dans le monde la 3d temps réel et sa rapide expansion dans les jeux vidéos par exemple. Ensuite pour le relatif challenge que le travail sur ce sujet constitue, en partie de part le manque de ressources inhérent à sa nouveauté. Et finalement pour l'intérêt que nous portons aux technologies de rendu bling-bling utilisées dans nos jeux préférés !

Comme vous le verrez plus tard, nous avons bien entendu focalisé notre travail sur le sujet principal de ce projet, mais nous nous sommes dit qu'il serait utile en même de toucher un peu à tout ce qui est abordé dans cette UV. Nous avons donc travaillés sur une multitude de points différents afin de mener à terme ce projet.

Reprécisons rapidement ce qu'est le HDR :

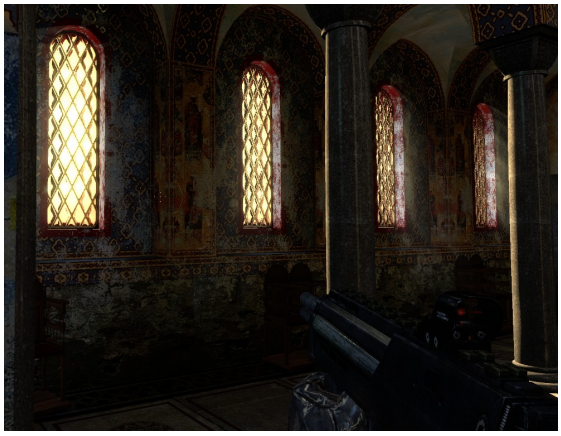
Le HDR (High Dynamic Range) est une technique permettant l'amélioration du rendu en essayant de s'approcher du fonctionnement de la rétine humaine capable de distinguer une plage d'intensités lumineuses largement supérieure à ce qu'une image standard RGB 24 bits est capable de stocker.

En effet, une image standard ne pourra avoir que 256 niveaux d'intensité lumineuse différents (8 bits, donc 2^8 niveaux) alors que l'oeil humain est capable d'en distinguer plusieurs dizaines de milliers.

L'idée principale est de trouver un moyen, et il y en a plusieurs valides, pour coder l'information de couleur sur plus de 24 bits et ainsi bénéficier d'une plage de valeur bien plus étendue, d'où le nom HDR. A partir de ce surplus d'information il est alors possible de réaliser de multiples effets, tel que simuler l'éblouissement ou contrôler l'exposition d'une scène, tel que le fait notre œil. Tout l'intérêt du HDR réside dans la conservation de valeurs supérieures à 1 pour une couleur.

Un exemple simple est celui du passage de la lumière à travers une fenêtre. La fenêtre va par nature absorber une partie de la lumière pour n'en restituer par exemple que 70%. Dans le cas du rendu standard (LDR), l'intensité d'une couleur ne pouvant dépasser 1, la lumière maximum en sortie de cette fenêtre sera (1, 1, 1) * 70%, une couleur gris clair donc. Le problème vient du fait que si c'est le soleil qui éclaire directement cette fenêtre, l'intensité étant bien plus élevé, disons pour l'exemple qu'il a une couleur valant (1000, 1000, 1000), la lumière en sortie devrait être bien supérieur à (1, 1, 1) et devrait valoir (700, 700, 700).

Half-Life 2: Lost Coast en LDR



Half-Life 2: Lost Coast en HDR

La puissance du HDR vient du fait que les valeurs ne sont tronquées qu'après le passage a travers la vitre, les reflets, etc… d'où en sortie une valeur de (700, 700, 700) du HDR, qui sera, à la fin seulement, tronquée à (1, 1, 1). Il est alors possible de simuler l'éblouissement via le blooming par exemple, technique que nous verrons plus tard dans ce rapport.

C'est l'opération de tone mapping, qui intervient en dernier et après l'application de l'ensemble des effets, qui permet d'arranger les valeurs HDR pour l'affichage sur un écran conventionnel, ayant donc des couleurs entre 0 et 1. Le tone mapping compresse l'ensemble de la plage de valeurs dans cet intervalle suivant certaines fonctions plus ou moins complexes.

Organisation du travail

Comme mentionné dans le cahier des charges nous avons dans un premier temps réalisé une phase d'investigation absolument fondamentale. Celle-ci a entre autre nécessité la lecture de plusieurs papiers mentionnés en bibliographie, mais aussi de présentations NVidia et ATI, et bien entendu de documentations officiels (GLSL, OpenGL…) et de portions de code dont nous nous sommes inspirées pour certaines parties pointues.

Il est à noter d'ailleurs que la littérature sur le sujet en OpenGL est très pauvre, et qu'il nous a souvent fallu nous tourner vers des exemples réalisés avec DirectX.

L'implémentation du logiciel a été faite de façon incrémentielle, en essayant de toujours avoir quelque chose de très stable et fonctionnel avant l'ajout d'une fonctionnalité nouvelle. Cette approche a beaucoup d'avantages, et notamment celui d'éviter de revenir en arrière trop souvent, mais prend du temps de part les multiples vérifications effectuées. Nous avons préféré développer un logiciel simple, robuste et relativement bien réalisé plutôt que de surenchérir dans le nombres d'options proposées au prix d'une perte de cohérence.

Nous avons donc dans un premier temps développé chacun des modules importants ; un template pour applications OpenGL avec la SDL, un module de chargement de shaders, des cameras… Nous avons ensuite basé tout le gros du projet sur ces classes, travaillant chacun des effets et shaders différents.

Comme vous vous en doutez, nous avons étés limité par le temps. Le sujet est en effet très vaste, et beaucoup de choses très intéressantes (et jolies) peuvent être faites sur la base de notre logiciel.

Vue d'ensemble

Nous avions décidé de développer un logiciel permettant d'expérimenter avec les techniques de rendus HDR, avec entre autre le blooming, simulant l'éblouissement provoqué par les zones de forte intensité lumineuse. Nous nous y sommes dans l'ensemble tenus.

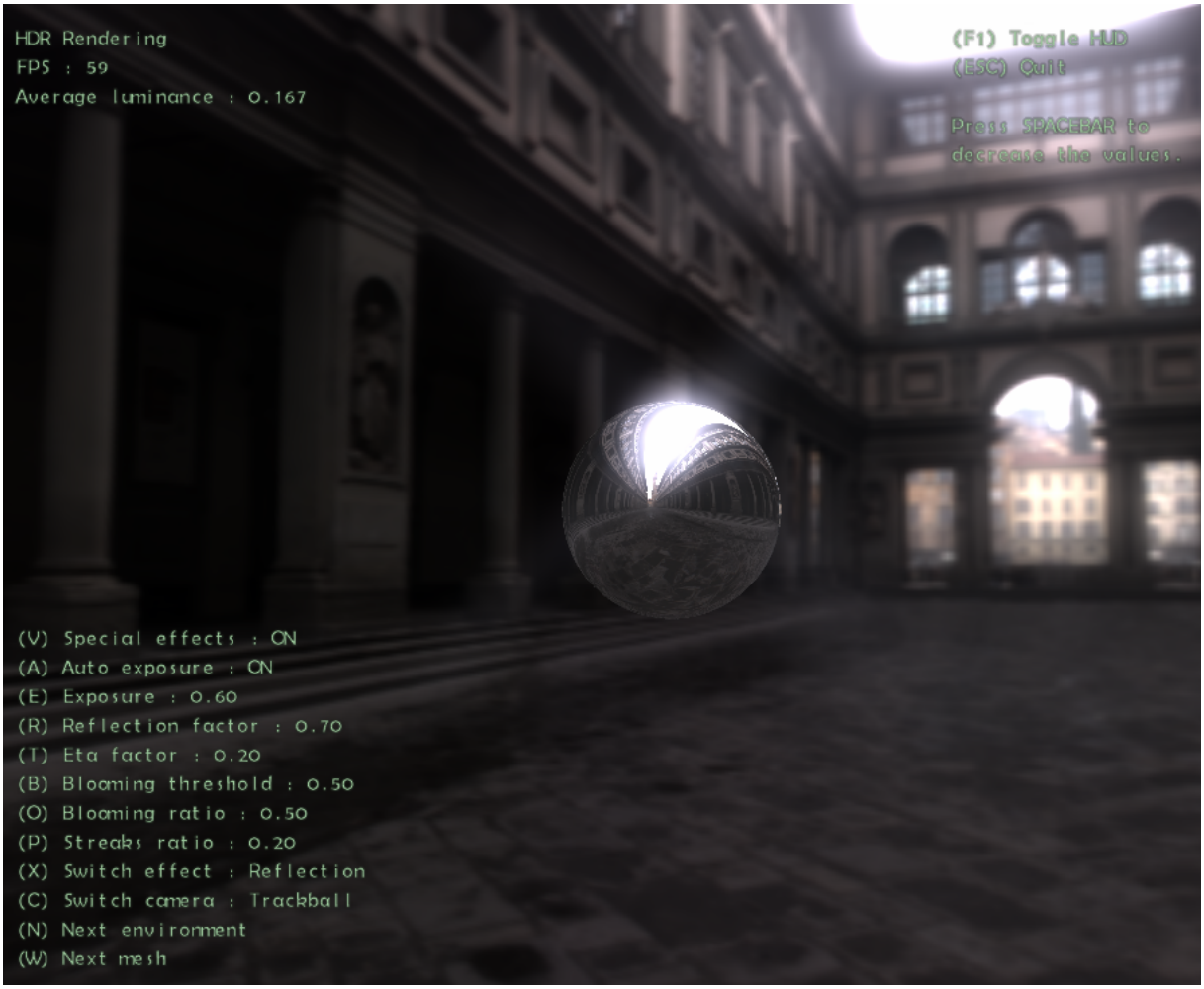
Comme précisé précédemment nous avons essayer autant que possible d'étendre notre travail à tout ce qu'il nous semblait indispensable d'avoir vu. Aussi nous avons dans un premier temps développé un support pour les programmes OpenGL avec la SDL. En ce qui concerne le choix de la SDL il est simple : nous avions déjà utilisé cette bibliothèque auparavant. L'idée de cette première partie était donc simplement de créer une surcouche objet pour la librairie SDL appliqué à OpenGL et la boucle de rendu.

Notre programme HDR vient se greffer sur ce code minimal nous permettant de créer simplement une application OpenGL, en surchargeant certaines classes abstraites comme nous le verrons un peu plus tard. L'application est alors simplement lancée comme suit.

```
int main(int argc, char *argv[])
{
    HDRApplication app("IN55 - HDR Rendering", 1024, 768);
    app.start();

    return 0;
}
```

L'application que nous avons développé sur cette brique de base met en évidence les bénéfices (et les problèmes) dues aux techniques de rendu HDR. Nous avons entre autre mis en œuvre le blooming et le tone mapping, deux techniques HDR, appliqué à un environnement HDR fournis sous forme de cube map. L'objet positionné au centre de la scène reflète ou réfracte son environnement. Etant lui-même au format HDR, on obtient alors des reflets dans ce même format sur lesquels nous pouvons appliquer différents effets.



Application au démarrage

Nous nous sommes inspirés du travail de Fabien Houlmann et Stéphane Metz pour mettre au point notre application, et leur travail a aidé à la recherche de documents grâce à leurs explications claires et la mise a disposition d'une bibliographie.

Les applications en terme de rendus sont dans l'ensemble assez ressemblantes et ce pour plusieurs raisons. Les map HDR utilisées pour effectuer le cube mapping d'environnement sont fournis par Paul Debevec, et utilisées partout dans le monde. Vous pourrez d'ailleurs les retrouver dans la plupart des démo portant sur le HDR. Nous les avons donc reprises. De même, nous avons repris quelques un des modèles utilisés dans leur application afin d'éviter des recherches trop fastidieuses.

Les techniques étant similaires, nous avons fait notre possible pour nous différencier de leur travail en implémentant en plus du blooming, indispensable, d'autres effets tels que l'ajustement automatique de l'exposition, semblable à ce que notre œil fait en présence de forte lumière, ou bien les streaks, des halos lumineux dues aux imperfections des matériaux sur lesquels la lumière se reflète.

L’utilisateur a la possibilité d’étudier ces effets séparément, suivant plusieurs paramètres contrôlable au clavier par une interface simple. Il pourra par exemple effectuer le rendu avec ou sans effets spéciaux, modifier la réflexion, la réfraction, étudier les effets de différents seuils pour les filtrages etc… Il aura aussi la possibilité de sélectionner différents environnements et maillages à afficher.

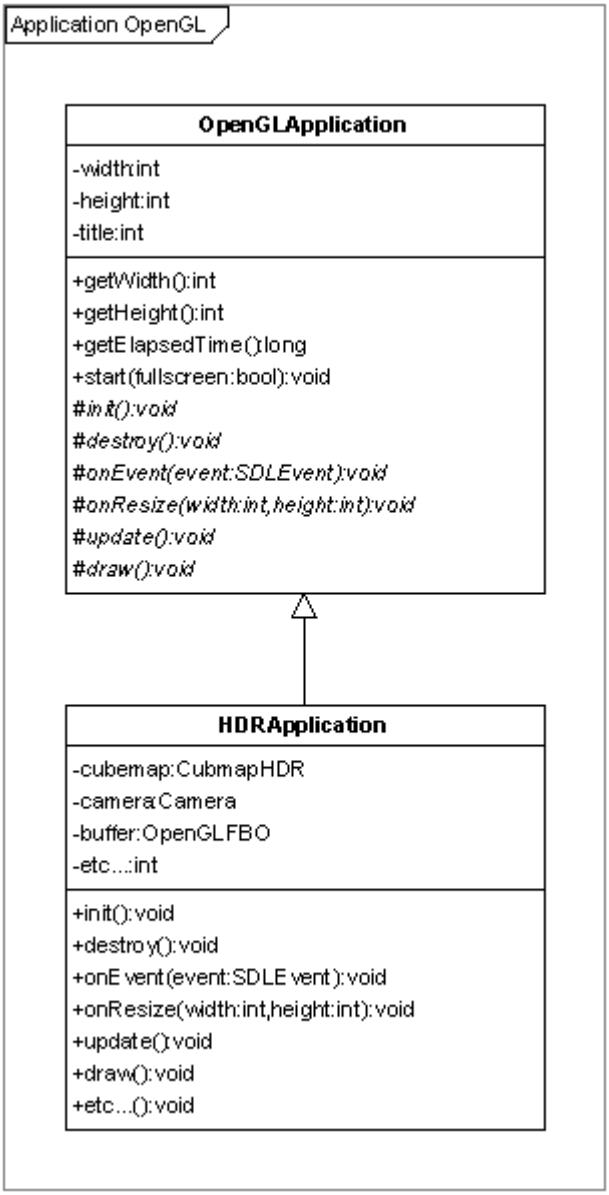
Notez que nous avons, dans notre quête du savoir, implémenté une mini bibliothèque permettant l’affichage de texte dans la fenêtre de rendu, sans l’utilisation d’outils externes tel que GLFont. Nous avons aussi mis au point deux types de camera contrôlées intégralement par quaternions.

Voyons maintenant en détail ce que ceci implique en terme d’implémentation.

Architecture et choix technologiques

Boucle de rendu et surcouche SDL

Nous avons adopté pour la base sur laquelle repose notre application un modèle similaire à celui que l'on peu trouver avec le framework XNA de Microsoft, ou avec le framework QT et sa boucle de rendu OpenGL.



Le principe est le suivant : la classe OpenGLApplication est abstraite. Elle contient plusieurs méthodes virtuelles pures et ne peu donc être instanciée. Il faut pour cela en créer une classe dérivée : HDRApplication dans notre cas, qui va spécifier le comportement à adopter pour chacune des situations abstraites.

La méthode fondamentale de la classe OpenGLApplication est la méthode start(…). Cette méthode va créer un contexte OpenGL et configurer la SDL dans ce but. Elle effectue alors un unique appel à la méthode init(…) de la classe fille puis rentre dans une boucle infinie depuis laquelle on va appeler les méthode onEvent(…), onResize(…), update(…) et draw(…).

Lors de la fermeture de la fenêtre elle effectue un appel à la méthode destroy(…) de la classe fille, afin de libérer les ressource personnalisées déclarées dans la classe dérivée, puis quitte l'application.

Les différents appels correspondent à :

- init(…) : appelée une unique fois au chargement de l'application.

- destroy(…) : appelée une unique fois à la fermeture de l'application.

- onResize(…) : appelée lorsque la fenêtre est redimensionnée, il faut alors effectuer les transformations nécessaires sur les buffers et viewport pour que la cohérence du rendu soit conservée.

- onEvent(…) : appelée à chaque tour de boucle et contient les événements SDL actuels, tel que les mouvements de souris, les appuis de touches etc… Il est alors possible de répondre à ces événements dans l'application fille.

- update(…) : appelée à chaque tour de boucle. C'est ici qu'il faut mettre à jour les données du programme, les positions des objet et des caméras, faire des calculs etc…

- draw(…) : appelée à chaque tour de boucle. C'est l'endroit ou il convient d'effectuer toutes les opérations de rendus OpenGL.

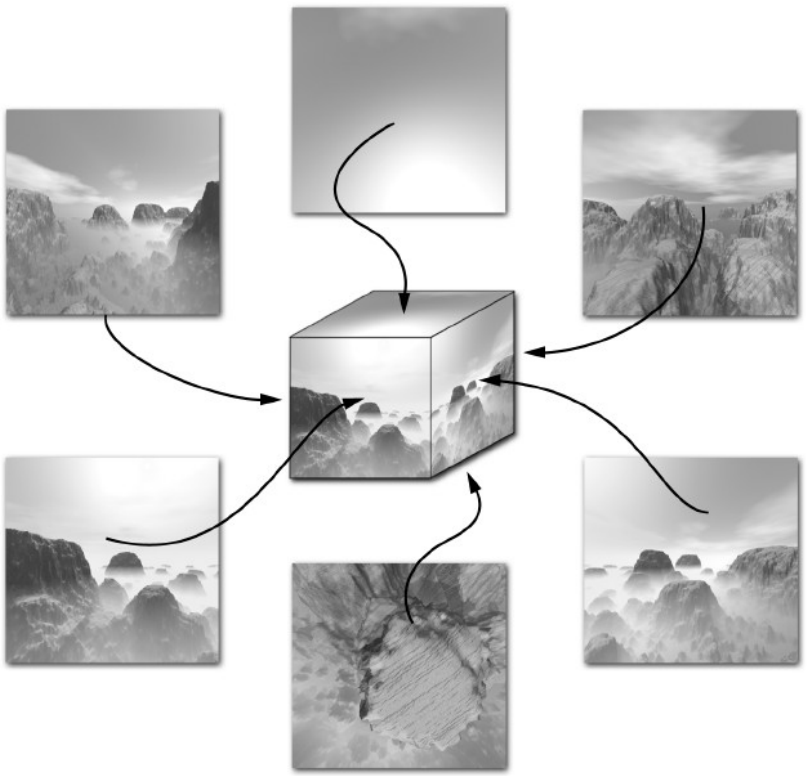
Techniques HDR

Avant toute chose, pour bénéficier du HDR, il convient de trouver une moyen de coder les couleurs sur plus de bits qu'elle n'en ont actuellement. Il existe différentes méthodes, en utilisant par exemple la composante alpha des couleurs comme un exposant, ou en utilisant des buffer spéciaux appelés FBO (frame buffer objects), similaires aux buffers OpenGL standards si ce n'est qu'il sont codés sur 32, 64 ou 128 bits. Nous avons dans le cadre de notre application utilisé des FBO de 64 bits.

Nous avons utilisé une classe nommé OpenGLFBO implémentant leur initialisation telle qu'elle est faite typiquement sur le net. Cette classe ne présentant pas grand-chose d'extraordinaire à part une longue suite d'initialisations est extraite du projet de Fabien Houlmann et Stéphane Metz que l'on remercie au passage pour l'avoir mis à disposition.

En terme de fichier HDR, pour les environnements, nous avons utilisé le format le plus populaire appelé Radiance pour lequel l'auteur a écrit en 1985 deux fichiers en C ANSI permettant leur chargement. Nous les avons réutilisés.

Voici les seules classes que nous avons réutilisées de part le web, si l'on met à part les bibliothèques SDL et GLEW pour le chargement des extensions bien entendu.
Nous avons a présent de quoi charger en ram des texture HDR, créer des cube map pour texturer notre environnement, et effectuer des rendus off-screen dans des FBO afin d'appliquer nos effets.



Cube mapping

Les techniques d'application d'effet sont a peu prêt toutes similaires. Le principe est d'effectuer un rendu non pas dans le frame buffer mais dans un autre buffer (un FBO par exemple) sur lequel il est possible d'effectuer différentes opérations avant l'affichage à l'écran. Celui-ci se comporte alors comme une texture 2d qu'il est possible d'appliquer sur un quad de la taille de la fenêtre pour effectuer le rendu final à l'écran.

Nous avons majoritairement utilisé des shaders pour mettre en place nos effets, car c'est en effet le moyen actuel le plus rapide et donnant les meilleurs résultats. Voyons comment réaliser les différents effets implémentés par notre application.

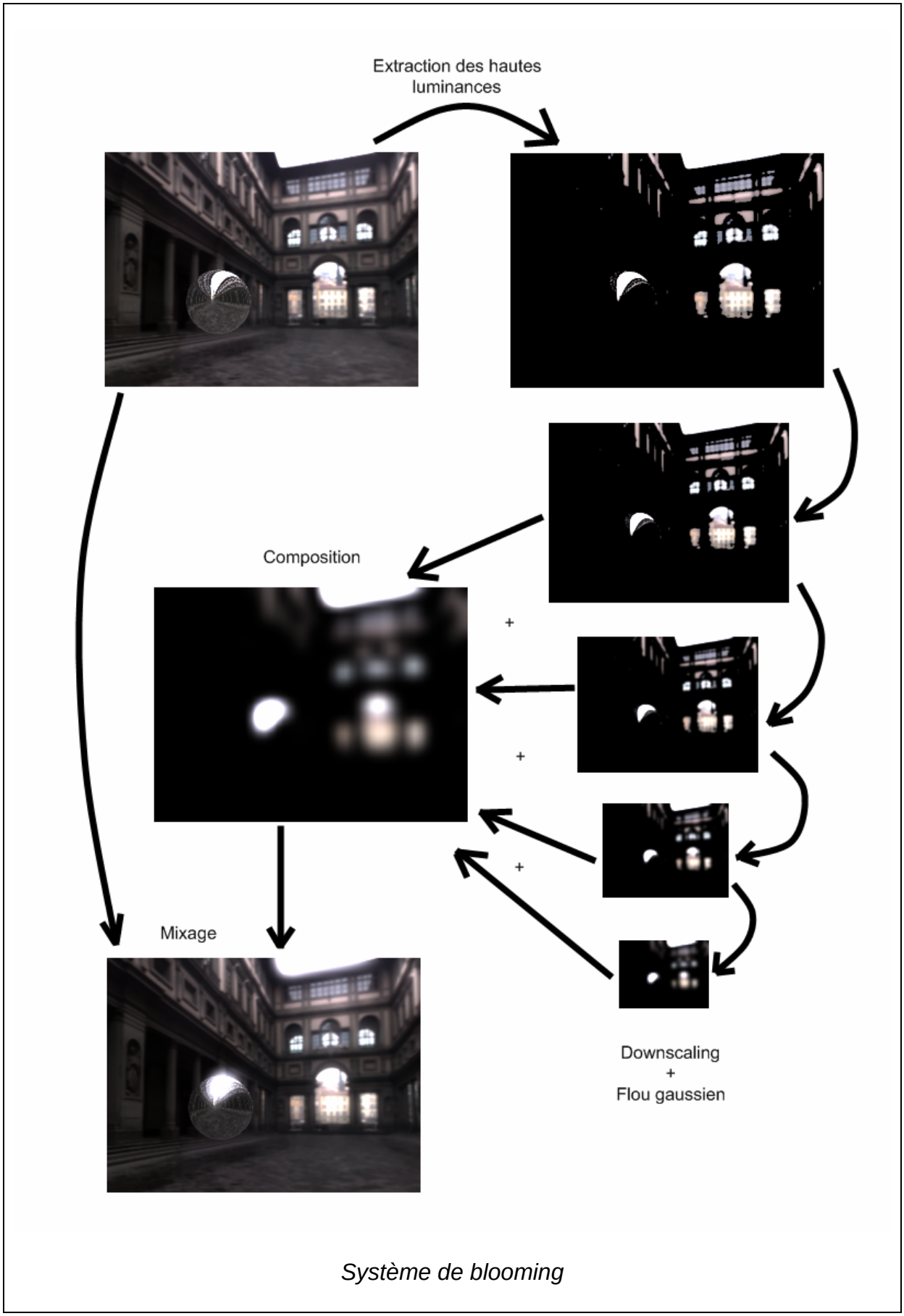
Blooming

Le blooming est un effet permettant de simuler l'éblouissement du a une source de forte intensité lumineuse. C'est cet effet de halo que l'on retrouve dans certains jeux récents, tel que Assassin's Creed, Far Cry et même plus anciennement Half-life2.

La réalisation d'un tel effet nécessite plusieurs passes d'application d'un filtre gaussien, puis sa composition avec l'imagine originale. L'idée est dans un premier temps d'extraire les zones à fortes intensités lumineuses pour en créer un masque. Ce masque est ensuite floué afin de faire « baver » les zones lumineuses sur les zones alentours. Dans l'idéal il faudrait appliquer un filtre gaussien 3x3 puis 5x5 etc … jusqu'à l'obtention du résultat escompté. Dans la pratique c'est impossible car un filtre supérieur à 5 est la cause de grosses pertes de performances. Il faut en effet appliquer une matrice de gauss 3x3, 5x5, etc… à chaque pixel de l'écran. On a donc par exemple un mixage entre 9 pixels alentours avec un filtre 3x3.

Afin de pouvoir bénéficier d'un effet correct sans atteindre trop les performances nous avons utilisé une techniques couramment utilisé pour ce genre d'effet qui consiste à appliquer le même filtre 3x3 à des texture downscalé, comprendre diminuées en taille par 2, 4, 8 etc… On filtre alors des textures de plus petites tailles, ce qui est beaucoup plus rapide, mais on floue aussi sur de plus grandes surfaces dans la mesure où la taille d'un pixel devient plus grande relativement à la taille de l'image.

Une fois le filtrage effectué on agrandi les textures en appliquant un filtrage bilinéaire (par défaut en OpenGL lors du placage), et après leur mixage on obtient un résultat proche de ce qu'un gros filtre aurait pu donner. Nous avons constaté qu'il était judicieux d'appliquer un filtrage gaussien lors du mixage afin d'estomper les traits résiduels dus aux textures les plus petites, particulièrement visible sur la démo de Fabien et Stéphane, mais ceci bien sur au prix d'une perte de performance.



Système de blooming

Notez qu'ici on effectue 4 passes pour le flou gaussien, ceci peu varier pour donner des résultats bien différents. En ce qui concerne l'opération de mixage, elle est en fait réalisée un peu plus loin dans le pipeline HDR mais elle aurait pu être faite ici. Nous verrons par la suite que l'on mixe tous les effets en même temps.

Streaks

Les streaks sont des effets qui s'apparente au blooming dans le sens ou on abuse a nouveau des flous. Un streak prend généralement la forme d'une croix. Cet effet est visible la nuit sur les phares de voitures. Il est principalement dû aux imperfections et égratignures sur les phares. Nous simulons simplement cet effet en mixant des pixels sur chacune des diagonales, dans les 4 directions.

Pour cet effet nous récupérons le résultat du blooming et y appliquons un shader. Ceci pour éviter d'avoir à filtrer l'image à nouveau. L'inconvénient est qu'il est alors dépendant des paramètres du blooming.

Masque de streaks

Voici le résultat de l'extraction des streak du masque de blooming de l'étape précédente. On obtient alors un nouveau masque que l'on va mixer lui aussi plus tard, en prenant soin de lui donner une intensité relativement faible pour que l'image soit la plus joli possible.

Si l'on augmente le seuil de luminance pour le blooming, les zones de hautes luminances étant beaucoup plus petites, les streaks le seront eux aussi. C'est un des problèmes inhérents à la réutilisation du masque de blooming.

Mixage

Une fois l'ensemble des effets générés il faut ensuite les combiner. Pour cette étape il existe une infinité de façons de faire. On peut les mélanger linéairement, logarithmiquement, … ou simplement les ajouter, résultant en une augmentation de la luminance moyenne de la scène.

Nous avons choisis de les ajouter à la scène d'origine, linéairement, avec un coefficient que l'utilisateur pourra définir. Il est alors possible visualiser les effet séparément en choisissant des coefficients égal à 0.

Exposition automatique

Au détour de nos recherches nous avons été impressionnés par une démonstration mettant en œuvre la correction d'exposition automatique. Nous avons-nous aussi implémenté une version de cette correction d'exposition automatique.

L'idée ici est de calculer la luminance moyenne de la scène et fonction de cette valeur modifier l'exposition de la scène en conséquence.

La formule est la suivante :

$$Lum_{avg} = \exp\left(\frac{1}{N}\sum_{x,y}\log(\delta + Lum(x,y))\right)$$

Afin de calculer la luminance moyenne rapidement il est indispensable de la calculer via les shaders. On constate ici que c'est l'exponentiel de la moyenne des logarithmes de la luminance de chaque pixel (avec delta une valeur petite dans le cas ou les pixels sont noirs, log(0) n'étant pas défini). Pour ce faire nous allons créer un rendu de la scène dans un buffer et assigner à la composante rouge de chaque pixel la valeur du log de la luminance. On a alors une texture contenant tous les log dont on veut faire la moyenne. Afin de faire ça le plus rapidement possible, on effectue un downscaling de la texture qui va moyenner le tout s'il y a filtrage, et ce jusqu'à obtenir une texture de 1x1 pixel. On obtient alors très rapidement le résultat.

Il est alors possible d'extraire l'unique pixel de cette texture, d'en calculer l'exponentiel, et calculer l'exposition correcte de la scène en utilisant la formule suivante :

$$Exp = \frac{Scale}{Lum_{avg}}$$

Pour trouver le cœfficient scale il n'y a pas d'autre moyen simple que de faire des essais de valeurs. On obtient alors une scène dont l'exposition varie instantanément

suivant les éléments du décor. Afin de simuler le temps d'adaptation de l'œil humain on souhaiterait établir une latence. Ceci se fait simplement par la formule suivante :

```
float exposure = 0.1 / m_lum;
m_exposure = m_exposure + (exposure - m_exposure)
                    * 0.002 * this->getElapsedTime();
```

Par cette formule on limite la variation d'exposition possible d'une image à l'autre, et ceci en fonction du temps pour ne pas la rendre dépendante du nombre de FPS de la machine. On obtient pour ces valeur un temps d'adaptation bien visible.

Tone mapping

Le tone mapping est l'étape finale du pipeline HDR. Il s'agit réajuster les valeurs des couleurs HDR pour qu'elles tiennent dans la page [0.0 – 1.0]. Il existe une multitude d'opérateurs différents qui ont fait l'objet de papiers longs et complexes sur lesquels nous sommes passés plus ou moins rapidement. Nous avons récupéré un opérateur simple prenant en compte l'exposition et l'avons implémenté dans un shader.

Le buffer contenant la scène à laquelle on a ajouté l'ensemble des effets est passé en paramètre à ce shader et le rendu final est alors généré.

Réflexion et réfraction

Afin que nos maillages reflètent l'environnement nous avons implémenté deux shaders standard de réflexion et de réfraction. Pour chaque vertex on calcul le rayon incident de l'œil, on le reflète ou réfracte sur la surface grâce à la normale du vertex, pour obtenir le rayon se dirigeant vers la texture d'environnement. Ce rayon est passé par une variable varying (donc interpolée) au pixel shader qui va, pour chacun des pixels, prendre la couleur du pixel de la cube map correspondant.

Démo HDR Rendering

L'application démarre sous Windows comme n'importe quelle autre application. Elle s'ouvrira en mode fenêtré 1024x768, redimensionnable à volonté. Si vous souhaitez en revanche la démarrer en mode plein écran il vous faudra la lancer depuis la console.

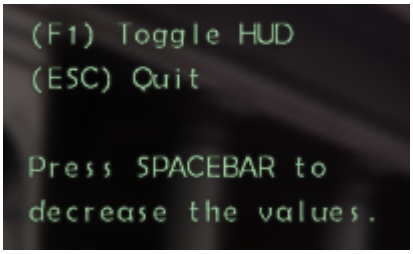
```
F:\Documents\UTBM\IN55\IN55_HDR\bin\Release>IN55_HDR.exe -fs1680x1050
```

En utilisant l'option –fs vous avez la possibilité de choisir entre plusieurs résolutions en plein écran. Il vous suffit de taper la commande précédente avec la résolution correspondante :

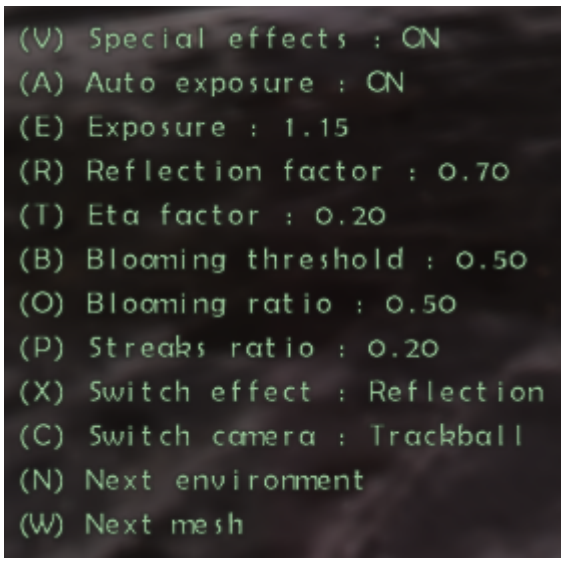
- 1680x1050
- 1280x1024
- 1280x768
- 1024x768
- 800x600
- 640x480

Une fois l'application lancée vous obtenez la fenêtre visible en début de rapport. Chacune des zones de texte vous donne des indications sur l'état courant du mode de rendu.

Cette boite contient le titre de la simulation et des informations non modifiables. Elle vous informe du nombre d'images rendues par seconde (bloqués à 60 à cause de la synchronisation verticale sur notre machine, beaucoup plus faible en salle de TP en revanche, et pas à cause de la VSync !). La seconde valeur est la luminance moyenne de la scène. C'est la valeur calculée pour modifier l'exposition automatiquement ; elle donne une idée de l'intensité lumineuse moyenne de l'image courante.



Une boite d'information vous indiquant qu'il est possible de cacher ce texte vert avec la touche F1, et de quitter l'application avec escape. Nous allons voir ensuite qu'il est possible de modifier des valeurs pour le rendu. Pour les décrémenter il vous faudra maintenir la barre d'espace simultanément.



La dernière et la plus importante des boites, elle vous donne les valeurs de tous les paramètres passés aux shader ou utilisées dans le rendu à un moment ou un autre. Entre parenthèse se trouve la touche à appuyer pour modifier la valeur. Utilisez espace simultanément pour décrémenter.

Touche	Nom	Effet
V	Special effects	Active ou désactive les effets spéciaux dans le rendu, c'est-à-dire le blooming, les streaks, et la manipulation de l'exposition.
A	Auto exposure	Active ou désactive la correction automatique d'exposition.
E	Exposure	Modifie l'exposition de la scène. Il est possible de la modifier en mode d'exposition automatique mais la valeur calculée continuera d'être ciblée par la correction d'exposition.
R	Reflection factor	Le facteur de réflectivité de la surface, soit la quantité de reflet par rapport à la couleur / texture de l'objet.
T	Eta factor	Le facteur propre a la réfraction définissant le type de matériel traversé par l'onde.
B	Blooming threshold	Le seuil de blooming, correspond à la valeur de luminance à partir de laquelle le blooming est appliqué. Avec une valeur de 0, le blooming est appliqué à l'ensemble de l'image.
O	Blooming ratio	La quantité de blooming rajouté à la scène. Avec une valeur de 0 le blooming disparaît.
P	Streak ratio	La quantité de streak rajouté à la scène. Avec une valeur de 0 les streak disparaissent.
X	Switch effet	Modifie l'effet appliqué au maillage : réflexion ou réfraction.
C	Switch camera	Modifie le type de camera utilisé : trackball ou freefly. La camera trackball correspond a une camera tournant autour de l'objet. Elle se contrôle via la souris uniquement : click droit pour le zoom, click gauche pour la rotation. La freefly est une camera libre. Elle se contrôle au clavier et à la souris : QSDZ pour avancer et straffer, click gauche pour l'orientation.
N	Next environment	Passe à la map d'environnement suivante. Il en existe un peu moins d'une dizaine.
W	Next mesh	Passe au maillage suivant. Il y en a 4 : une sphère, le teapot, un visage, une tasse. Notez que les modèles sont compilés dans l'exécutables et que la sphère et un quadric.

Bibliographie

La bibliographie est volontairement bien fournie, pour la bonne et simple raison qu'il est relativement compliqué de trouver rapidement des informations intéressantes (en terme de programmation) sur ce sujet. Vous aurez donc ici même l'ensemble des documents nécessaires à la réalisation d'une telle application.

Cube mapping :
http://developer.nvidia.com/object/cube_map_ogl_tutorial.html
http://texel3d.free.fr/opengl/cubemapping/index.html

Format Radiance RGBE :
http://www.graphics.cornell.edu/%7Ebjw/rgbe/
http://www.debevec.org/Probes/

GLSL :
http://www.lighthouse3d.com/opengl/glsl/index.php?intro
http://nehe.gamedev.net/data/articles/article.asp?article=21
http://www.opengl.org/sdk/libs/OpenSceneGraph/glsl_quickref.pdf

Effets de rendu :
http://ati.amd.com/developer/gdc/gdc2003_scenepostprocessing.pdf
http://www.courseptr.com/downloads/chapterpreview/00924-ch8prev.pdf
http://msdn.microsoft.com/en-us/library/bb173484(VS.85).aspx
http://www.cse.ohio-state.edu/~kerwin/refraction.html
http://tams-www.informatik.uni-hamburg.de/applets/hades/webdemos/00-intro/02-imageprocessing/gauss.html

Tone mapping :
http://en.wikipedia.org/wiki/Tone_mapping
http://www.cs.utah.edu/~reinhard/cdrom/tonemap.pdf
http://graphics.cs.yale.edu/publications/egwr2000_tonemapping.pdf
http://fr.wikipedia.org/wiki/Luminance
http://luxal.dachary.org/webhdr/tonemapping.shtml#glob
http://www.mpi-inf.mpg.de/resources/tmo/

Généralités HDR et démos :
http://www.daionet.gr.jp/~masa/rthdribl/
http://www.labri.fr/perso/pouderou/SI/exposes/2005-2006/hdr.pdf
http://www.gamedev.net/reference/articles/article2208.asp
http://www.easyhdr.com/tutorial.php?sub=4#s